# GD2S03
# Advanced Software Engineering & Programming for Games

MEDIA DESIGN SCHOOL

Bachelor of Software Engineering(BSE)
Game Development

- Overview
  - Nested Types
  - Extensions
  - Protocols

# Nested Types

- To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required

```
struct BlackjackCard {
// nested Suit enumeration
  enum Suit: Character {
  case spades = "♠",hearts = "♡",diamonds = "♢",clubs = "♣"
  }
// nested Rank enumeration
  enum Rank: Int {
      case two = 2, three,four,five,six,seven,eight,nine,ten
      case jack, queen, king, ace
      struct Values {
          let first: Int, second: Int?
      }
      var values: Values {
         switch self {
           case .ace:
             return Values(first: 1, second: 11)
           case .jack, .queen, .king:
             return Values(first: 10, second: nil)
           default:
             return Values(first:self.rawValue,second:nil)
         }
      }
  }
```

```
// BlackjackCard properties and methods
  let rank: Rank, suit: Suit
  var description: String {
     var output = "suit is \(suit.rawValue),"
     output += " value is \(rank.values.first)"
     If let second = rank.values.second {
        output += " or \(second)"
     }
     return output
   }
}
let aceOfSpades = BlackjackCard(rank: .ace,
                                suit: .spades)
print("aceOfSpades:\(aceOfSpades.description)")
//Prints "aceOfSpades: suit is ♠,value is 1 or 11"
```

To use a nested type outside of its definition context, prefix its name with the name of the type it is nested within:

```
let hearts = BlackjackCard.Suit.hearts.rawValue
// hearts is "♡"
```

# Extensions

- *Extensions* add new functionality to an existing class, structure, enumeration, or protocol type.

- This includes the ability to extend types for which there is no access to the original source code.

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

- Declare extensions with the ***extension*** keyword:

- An extension can extend an existing type to make it adopt one or more protocols.

- To add protocol conformance, write the protocol names the same way as it's written for a class or structure:

```
extension SomeType {
// new functionality to add to SomeType goes here
}

extension SomeType: SomeProtocol, AnotherProtocol {
// implementation of protocol requirements goes here
}
```

# Extensions – Computed Properties And Initializers

- Extensions can add computed instance properties and computed type properties to existing types

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters“

let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

# Extension - Initializers

- Extensions can add new initializers to existing types.
- Extensions can add new convenience initializers to a class.
- Extensions cannot add new designated initializers or deinitializers to a class.
- Designated initializers and deinitializers must always be provided by the original class implementation.

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}

let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x:
2.0, y: 2.0),
      size: Sze(width: 5.0, height: 5.0))
```

- Rect structure can be extended to provide new initializer that takes specific center point and size

```
extension Rect {
  init(center: Point, size: Size) {
    let originX = center.x - (size.width / 2)
    let originY = center.y - (size.height / 2)
    self.init(origin: Point(x: originX,
                            y: originY), size: size)
  }
}

let centerRect = Rect(center: Point(x:4.0,y:4.0),
                 size: Size(width:3.0,height:3.0))
// centerRect's origin is (2.5, 2.5) and its size
is (3.0, 3.0)
```

# Extension - Methods

- Extensions can add new instance methods and type methods to existing types.
- The following example adds a new instance method called **repetitions** to the **Int** type:

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}

3.repetitions {
    print("Hello!")
}

// Hello!
// Hello!
// Hello!
```

- Instance methods added with an extension can also modify (or *mutate*) the instance itself.
- Structure and enumeration methods that modify **self** or its properties must mark the instance method as **mutating**, just like mutating methods from an original implementation.

```
extension Int {
    mutating func square() {
        self = self * self
    }
}

var someInt = 3
someInt.square()
// someInt is now 9
```

# Extension – Subscripts and Nested Types

- Extensions can add new subscripts to an existing type.
- This example adds an integer subscript to Swift's built-in **Int** type

```
extension Int {
  subscript(digitIndex: Int) ->
Int {
    var decimalBase = 1
      for _ in 0..<digitIndex {
        decimalBase *= 10
      }
    return(self/decimalBase)%10
  }
}

746381295[0] // returns 5
746381295[1] // returns 9
746381295[2] // returns 2
746381295[8] // returns 7
```

- Extensions can add new nested types to existing classes, structures, and enumerations:

```
extension Int {
  enum Kind {
    case negative, zero,
        positive
  }
  var kind: Kind {
    switch self {
    case 0:
      return .zero
    case let x where x > 0:
      return .positive
    default:
      return .negative
    }
  }
}
```

The nested enumeration called Kind, expresses the kind of number that a particular integer represents

```
func printIntegerKinds(_ numbers: [Int]) {
  for number in numbers {
    switch number.kind {
    case .negative:
      print("- ", terminator: "")
    case .zero:
      print("0 ", terminator: "")
    case .positive:
      print("+ ", terminator: "")
    }
  }
  print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + - 0 - 0 + "
```

- This function, takes an input array of Int values and for each integer in the array, the function considers the kind computed property and prints an appropriate description.

# Protocols

- A *protocol* defines a blueprint of methods, properties, and other requirements.
- A class, structure, or enumeration *adopts* protocol and *conform* to that protocol by providing an actual implementation of those requirements.

- Protocols are defined in a similar way as classes, structures or enumerations.
- If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

## Property Requirements

- Protocol provides the name and type of a property.
- Property requirements are always declared as variable properties, prefixed with the *var* keyword
- Protocol  specifies whether the property must be *gettable* or *gettable* and *settable but* doesn't specify whether the property should be stored or computed.
- Always prefix type property requirements with the static keyword

```
protocol SomeProtocol {
// protocol definition goes here
}

class SomeClass: SomeSuperclass,
FirstProtocol, AnotherProtocol {
// class definition goes here
}

protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int {
get }
}

protocol AnotherProtocol {
    static var someTypeProperty: Int {
get set }
}
```

# Protocol - Requirements

## Method Requirements

- Methods are written as part of protocol's definition.
- Curly braces or a method body is not allowed.
- Variadic parameters are allowed.
- Default values can't be specified within protocol's definition.
- Type methods are prefixed with *static* keyword.

```
protocol SomeProtocol {
    static func someTypeMethod()
}
protocol RandomNumberGenerator {
    func random() -> Double
}
```

## Mutating Method Requirements

- If a protocol's method requirement is to mutate instances of any type that adopts the protocol, the method is marked with ***mutating*** keyword.

```
protocol Togglable {
    mutating func toggle()
}
```

## Initializer Requirements

- Initializers in protocol are written without any curly braces.
- A conforming class can implements a protocol's initializer as either designated or convenience initializer.
- The use of the ***required*** modifier ensures that the subclasses of the conforming class, also conform to the protocol.
- A ***final*** class must not use ***required*** modifier.

```
protocol SomeProtocol {
    init(someParameter: Int)
}

class SomeClass: SomeProtocol {
    required init(someParameter:
Int) {
  //initializer implementation here
    }
}
```

# Protocol - Requirements

- If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the **required** and **override** modifiers

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
    // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    //"required" from SomeProtocol conformance;
      "override" from SomeSuperClass
    required override init() {
     //initializer implementation goes here
    }
}
```

Failable Initializer Requirements

- Protocols can define failable initializer requirements for conforming types
- A failable initializer requirements can be satisfied by a failable or nonfailable initializer on a conforming type.
- A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer

# Protocols as Types

Protocols can be used as a type and can be used in may places where types are allowed, including

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

```
protocol RandomNumberGenerator {
    func random() -> Double
}
class LinearCongruentialGenerator:
                        RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a +
c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
```

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator:
RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() *
Double(sides)) + 1
    }
}
var d6 = Dice(sides: 6, generator:
LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
```

# Protocols - Delegation

- *Delegation* is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type.
- This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated.
- Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

```
protocol Cook: class{
    func cookFood(for object: String)
}
class Chef: Cook{
    func cookFood(for object:String) {
        print("Cooking Food for \(object)") }
}
class Mom: Cook{
    func cookFood(for object:String) {
        print("Cooking Homemade Food for \(object)") }
}
class JustEater{
    var delegate:Cook
    init(delegate: Cook){
        self.delegate = delegate
    }
    func wantToEat(){
        delegate.cookFood(for: "JustEater") }
}
let justEater = JustEater(delegate: Chef())
justEater.wantToEat()

justEater.delegate = Mom()
justEater.wantToEat()
```

# Protocol – Adding Conformance with an Extension

- An existing type can be extended to adopt and conform to a new protocol even if the source code for that type is not accessible.

```
protocol DontCook{
    func cantCookFood()
}
protocol Cook{
    func makeFood()
}
class Chef: Cook{
    func makeFood() {
        print("Cooking")}
}
let chef = Chef()
chef.makeFood()

extension Chef: DontCook{
    func cantCookFood() {
        print("can't cook")}
}
chef.cantCookFood()
```

Conditionally Conforming to a Protocol

```
protocol Cook{
    var makeFood: String {get}
}

class Mom: Cook{
    var foodName: String
    init(_ foodName: String){
        self.foodName = foodName
    }
    var makeFood:String {
     return "Cooking \(foodName)"}
}

extension Array where Element: Cook{
    func printItems(){
        for item in self{
           print(item.makeFood)}
    }
}
let cook = [Mom("burger"),
Mom("pizza"), Mom("fries")]
cook.printItems()
```

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
protocol Cook{
    func makeFood()
}

class Mom{
    func makeFood() {
        print("Cooking")
    }
}
extension Mom: Cook{}
```

# Protocol – Protocol Inheritance

- A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits.

- The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

- In the example on the right, anything that adopts **CookVeryNiceFood** must satisfy all of the requirements enforced by **Cook**, *plus* the additional requirements enforced by **CookVeryNiceFood**.

```
protocol Cook{
    var makeFood: String {get}
}

protocol CookVeryNiceFood: Cook{
    var makeVeryNiceFood: String { get}
}

class Mom: CookVeryNiceFood{
    var foodName: String
    init(_ foodName: String){
        self.foodName = foodName
    }
    var makeVeryNiceFood: String{
        return "cooking very nice \(foodName)"}

    var makeFood: String{
        return "cooking \(foodName)"}
}

let mom = Mom("pizza")
print(mom.makeVeryNiceFood)
```

# Protocols – Composition

- Multiple protocols can be combined into a single requirement with composition.
- Protocol composition doesn't define any new Protocol types.
- Protocol composition have the form of *SomeProtocol & AnotherProtocol*.
- Any number of protocols cane be listed together with ampersends *(&)*.
- Composition can also be made between class and a protocol

```
protocol Named {
  var name: String { get }
}
protocol Aged
  var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}

func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name),
    you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

```
protocol Named{
    var name: String {get}
}
class Aged{
    var Age: Int
    init(_ age: Int){
        self.Age = age;
    }
}

class Person: Aged, Named  {
    var name: String
    init(name: String, Age: Int){
        self.name = name
        super.init(Age)
    }
}
func wishHappyBirthday(to celebrator: Named & Aged){
    print("Happy Birthday, \(celebrator.name),
\(celebrator.Age)")
}
let birthdayPerson = Person(name: "Malcolm", Age: 21)
wishHappyBirthday(to: birthdayPerson)
```

# Protocol – Checking for Conformance

- ***is*** and ***as*** operators is used to check for protocol conformance, and to cast to a specific protocol.
- Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:
- The is operator returns true if an instance conforms to a protocol and returns false if it doesn't.
- The as? version of the downcast operator returns an optional value of the protocol's type, and this value is nil if the instance doesn't conform to that protocol.
- The as! version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn't succeed.

```
protocol HasArea{
    var description: String {get}
}

class Circle: HasArea{
    var description: String{ return "Circle has Area” }
}

class Square: HasArea{
    var description: String{ return "Square has Area” }
}

class Animal{
    var description: String{ return "Kind of Animal” }
}

let object: [AnyObject] = [Circle(), Square(),
Animal()]
for item in object{
    if let objWithArea = item as? HasArea{
        print(objWithArea.description)
    }
    else{ print("complex Area") }
}
```

# Protocol – Optional Requirements and Constraints

- *optional requirements* can be defined for protocols.
- These requirements don't have to be implemented by types that conform to the protocol.
- Optional requirements are prefixed by the **optional** modifier as part of the protocol's definition.
- Optional requirements are available so that you can write code that interoperates with Objective-C.
- Both the protocol and the optional requirement must be marked with the **@objc** attribute.
- Note that **@objc** protocols can be adopted only by classes that inherit from Objective-C classes or other **@objc**classes.
- They can't be adopted by structures or enumerations.

```
@objc protocol MyProtocol {
    @objc optional func doSomething()
}

class MyClass : MyProtocol{

    // no error
}
```

- A protocol extension, can be specified with constraints that conforming types must satisfy before the methods and properties of the extension are available.
- These constraints after the name of the protocol being extended using a generic where clause

```
extension Collection where Element:
Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

# Protocol - Extension

- Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types.

- Protocol extension can be used to provide a default implementation to any method or computed property requirement of that protocol.

```
protocol RandomNumberGenerator{
    func random()->Double
    func RandomBool()->Bool{
}

extension RandomNumberGenerator{
    func RandomBool()->Bool{
        return random() > 0.5
    }
    func defaultRandom(random: UInt32)->Double{
       return Double(arc4random_uniform(random))
    }
}

class GenerateRandomNumber: RandomNumberGenerator{
    func random() -> Double {
        return Double(arc4random_uniform(10))
    }
}

let generateRandomNumber = GenerateRandomNumber()
print(generateRandomNumber.defaultRandom(random: 20))
```